low level virtual machine

编译器. 工具. 框架. 工具链.

▲ 优化以任意程序语言编写的程序的
- compile time
- link-time
- run time
- idle time

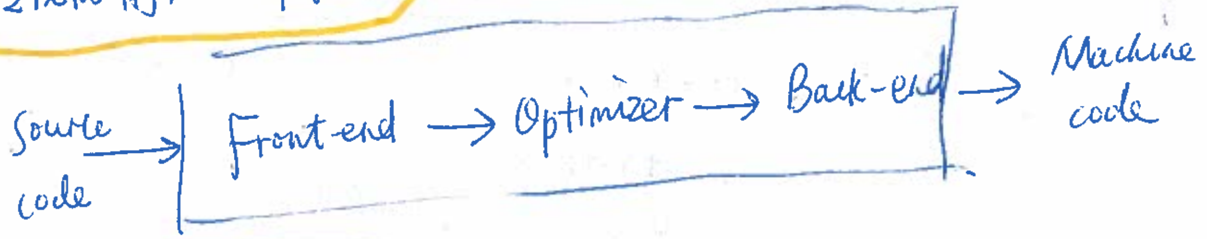
▲ 高度模块化.

与语言无关的中间代码

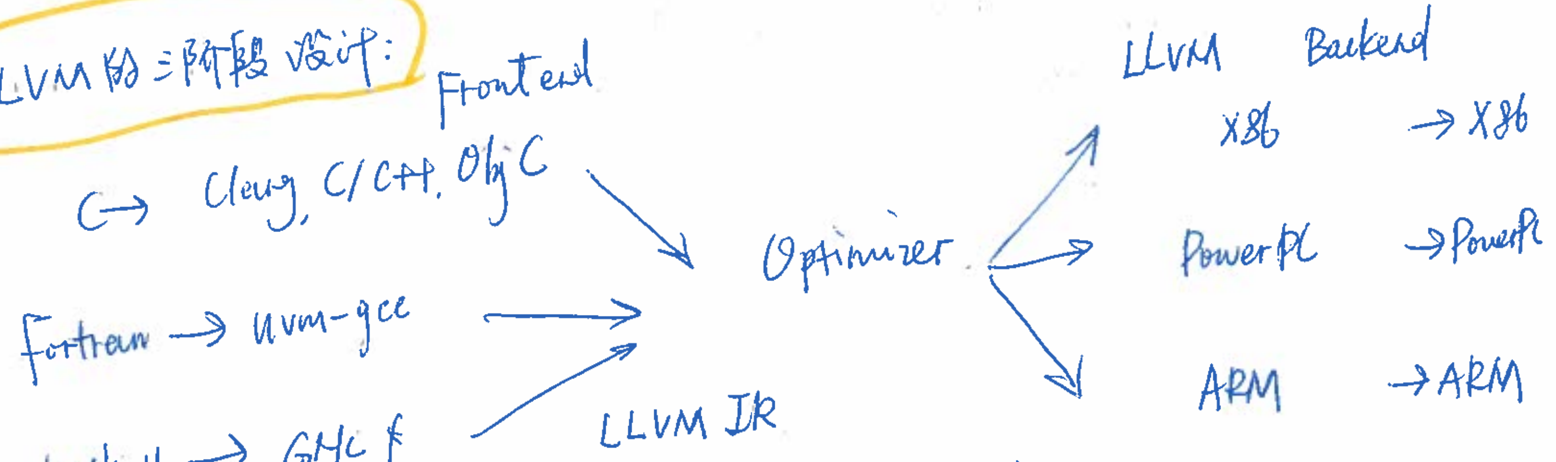
将不同语言相互连结起来.

能够紧密地与IDE交互和集成.

传统的静态编译器:



LLVM的三阶段设计:



→优化阶段针对统一的 LLVM IR 是通用的.

↓ 如果需要支持一种新的语言. 则只需实现一种新的前端

↓ 只需要支持一种新的硬件设备 则只需要实现一个新的后端

LLVM IR 通常有三种格式

（一般无法直接读取到）

= { 在内存中的编译中间语言
硬盘上存储的二进制中间语言（以 .bc 结尾）
可读的中间格式（以 .ll 结尾）

是 LLVM 优化和进行代码生成的关键.

根据可读的IR → know：在最终生成目标代码前，我们已生成了怎样的代码.
→ 选择使用不同后端来生成不同的代码.

A compiler called Clang that takes C++/其他语言 and translate it to a simplified program representation (LLVM IR)

a target-independent backend that performs various analyses and transformations on LLVM IR "optimizer"

a target-dependent backend that lowers LLVM IR to native machine code like x86.

LLVM IR ↓ An assembly language

contains types, such as i32, i8*

has unlimited number of registers with names that start with %

寄存器

No register occurs on the left hand side of more than one assignment.

指针用 →
非指针用点.

LLVM Optimizer.

↓

Analyzes, optimizes, secures programs.

↓

operates on LLVM Intermediate Representation (IR) code.

⟹ source and target-independent.

↓

Functionalities are implemented as Passes.

↓

A pass is an operation on a unit of LLVM intermediate representation (IR) code

LLVM IR code is:

A low-level
Strongly-typed
language-independent,
SSA-based representation.

↓

Tailored for Static analyses
& optimization purposes.

ModulePass. CallGraphPass. FunctionPass LoopPass.
RegionPass, BasicBlockPass.

↓ Hierarchy:

Module → == A compilation unit.
(code and code included)

↓

Function

↓

Basic Block

↓

Instruction.

5. project — part 1 作业时.

先. cd → CSE231_project folder.

sudo ./mount_and_launch.sh 把各个文件 mount在一起.

这个 script will automatically mount the other three folders to the appropriate mount points in the docker image.

→ 所以这个时候就进入了 LLVM的 folder.

↓

use docker to compile and run

LLVM folder

/LLVM_ROOT / llvm source code

/ build compiled LLVM

Passes → /CSE231_project folder

Tests → /tests

Output → / output.

```
clang++ -c -O0 -emit-llvm /lib231/lib231.cpp
        -S
```

小c → get .bc file

大S → get .ll file

把 c/cpp/... 文件编译成 IR code (assembly code)

cd to Passes folder. make

make所做的事情:

第一个 CMakeLists 个 sub_directory < testPass / part 1.

testPass里 CMakeLists → 生成名字叫 LLVMTestPass.so

~~里面~~ 是通过 TestPass.cpp生成的.

part 1里 CMakeLists → 生成 CSE231.so

里面通过 3个 cpp 生成.

每个cpp都是一个pass 有自己的名字.

我们在写 passes时. 应该用错误流输出. 而不是通过标准输出流输出.

errs( )　　　　outs( )

在 make 之后　run your pass.

~~Section 1 Static Inst.~~ Testpass:

输入　　输出

```
opt -load LLVMTestPass.so -TestPass < ....ll > /dev/null
```

该so名字

该pass名字

不需输出了.

LLVM的command line. 用以 execute passes

load 含有你需要的那个 pass的这个....so library.

2> ....txt

输出成txt文件.

因为在写pass的cpp文件 后面 RegisterPass <name>

实际在 section 中

先 cd /tests/test-example

(cuz test file在这里)

```
opt -load /LLVM_ROOT/build/lib/CSE231.so -cse231-bb
    < test1.ll -o test1-instrumented.ll
```

original version

生成

新文件 instrumented version

对 clang++ lib231.bc test1-instrumented.bc ~~-o mytest~~ test1-main.cpp -o test1

1　2　3

executable file.

link 1 runtime library 2 instrumented bitcode 3 → linked program

In LLVM.

each pass → implemented in → separate C++ class.

↓

Pass class → FunctionPass ..... ModulePass. BasicBlockPass.

↓ TestPass.

ModulePass ↓ A single module at a time

BasicBlockPass ↓ A Basic Block at a time

FunctionPass ↓ implements functionality for analyses and optimizations that only look at a single function at a time.

For each kind of Pass. there is a corresponding entry point function runOn<Suffix>

Suffix ↓ the kind of Pass

e.g. runOnFunction (Function &F) {..}.

this will be called for each function in the program.